EV355228745

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

CONVERSION OF STRUCTURED DOCUMENTS

Inventors:

Laurent Mollicone

Andrew P. Begun

Ned B. Friend

and

Stephen J. Mooney

ATTORNEY'S DOCKET NO. MS1-1556US

TECHNICAL FIELD

This invention relates to upgrading documents so that the documents are compatible with a version of a document processing mechanism that is used to process the documents. In a more particular implementation, this invention relates to upgrading arbitrary markup language documents so that the documents match a version of a document processing mechanism used to display and edit the markup language documents. In this disclosure, the term "upgrading" has broad connotation, encompassing any kind of modification of a document.

BACKGROUND

A document created using a particular version of a document processing mechanism often cannot be satisfactorily processed by later versions of the document processing mechanism. For example, documents created using a particular version of a word processing application or a spreadsheet application (referred to as "original documents") often cannot be adequately processed by later versions of these applications. For instance, the original documents may lack information that is needed to fully exploit enhanced functionality provided by the later developed versions of the applications. This can result in the suboptimal rendering of the original documents in the later developed versions of these applications, or in extreme cases, the inability to render any information gleaned from the original documents. And even if the original documents can be displayed, these documents may exhibit suboptimal behavior when processed by later versions of these applications.

Applications developed specifically to render and process markup language documents share the above shortcomings. A typical application includes an Extensible Stylesheet Language (XSL) processor that transforms a document expressed in the

Extensible Markup Language (XML) to a document expressed in some presentation-oriented markup language, such as Hypertext Markup Language (HTML). The XSL processor uses a collection of XSL files in transforming XML into HTML, wherein these files effectively specify the appearance of the document as rendered using HTML. The XSL files might have been specifically developed to process particular kinds of XML documents characterized by a specified schema. Subsequently, a developer may have made significant changes in the XSL files to enable processing of new kinds of XML documents having new characteristics, possibly governed by a new schema. Due to these changes, the XSL processor might not be able to satisfactorily process the kinds of XML documents for which it was originally designed. The presentation of the original XML documents using the upgraded XSL files may produce errors, or may be completely proscribed.

Fig. 1 illustrates a conventional strategy for addressing the above-noted problems. Fig. 1 specifically addresses the case of a program-oriented application, such as a word processor (rather than a declarative-oriented rendering mechanism, such as a markup language browser). In this environment, an application program is upgraded from an application version V1 102 to an application version V2 104. Application version V2 104 may include additional functionality compared to application version V1 102, or may omit certain functionality present in the application version V1 102.

In the scenario shown in Fig. 1, a document 106 has been generated by application version V1 102, and it is subsequently desired to process this document 106 using application version V2 104 and render this document 106 on a display device 108. The conventional strategy for handling this task is to provide conversion logic 110. The conversion logic 110 converts the document 106 into a form suitable for processing by application version V2 104 and for presentation on the display device 108. The

conversion logic 110 can be implemented as add-on code associated with application version V2 104.

The above strategy has a number of drawbacks. The conversion logic 110 is specifically tailored to translate information produced by application version V1 102 into information expected by application version V2. Hence, the characteristics of the conversion logic 110 are specifically predicated on a detailed comparison between application versions V1 and V2 (102, 104). As such, the conversion logic 110 might not be able to satisfactorily process documents produced by other sources, and it might not be able to process documents in conjunction with other versions of document processing mechanisms. For example, the conversion logic 110 might not be able to process a document produced by some other predecessor version of the application, such as an application version V0.5, etc. Further, the conversion logic 110 may no longer provide satisfactory results for documents 106 produced by application version V1 102 when the application program is upgraded in the future to a later version – say, for example, version V3. In summary, the conversion logic 110 cannot, and was never intended to, handle documents having arbitrary form and content. In other words, the conversion logic 110 is narrowly tailored to the task of translating between applications V1 102 and V2 104. Markup language processors (e.g., browsers) often resort to a similar tactic to that shown in Fig. 1, and therefore suffer from the same shortcomings discussed above.

In the traditional approach, a developer would address these problems by adding modules to the conversion logic 110 to address different permutations in document conversion scenarios. This has drawbacks, because it requires the developers to develop new code each time the processing environment changes.

Based on the foregoing, there is an exemplary need in the art for a more efficient and flexible technique for upgrading documents so that they are compatible with current

versions of document processing mechanisms, such as markup language document processing mechanisms.

SUMMARY

According to one exemplary implementation, a method is described for upgrading documents for processing by processing functionality. The method includes: (a) inputting a structured document having particular features associated therewith into a particular version of the processing functionality; (b) determining whether each of the particular features matches a set of expected features associated with the particular version of the processing functionality; and (c) modifying the particular features of the input structured document so that the particular features match the set of expected features to thereby provide a modified structured document. After the above-described modification, the method includes: (d) transforming the modified structured document into another document suitable for presentation; (e) displaying the other document suitable for presentation using the processing functionality to provide a displayed document; and (f) editing the displayed document.

The input structured document can be expressed in the extensible markup language (XML). The other document can be expressed in the hypertext markup language (HTML). In one implementation, the operation of modifying is implemented using an upgrade module that provides a transformation function using extensible stylesheet language (XSL).

The above-referenced determining of whether each of the particular features matches a set of expected features associated with the particular version of the processing functionality can include determining whether the input structured document contains each node expected by the particular version of the processing functionality. The above-

referenced modifying of the particular features of the input structured document to produce the modified structured document can include: (c1) creating each node expected by the particular version of the processing functionality to provide created nodes; (c2) copying node content from the input structured document into corresponding created nodes in the modified structured document for those nodes in the input structured document that have counterpart nodes expected by the particular version of the processing functionality; and (c3) creating default node content in corresponding nodes in the modified structured document for those created nodes that do not have counterpart nodes in the input structured document.

In one case, the above-referenced "expected features" are specified by a schema associated with the particular version of the processing functionality. In another case, the above-referenced "expected features" are specified by some information other than the schema associated with the particular version of the processing functionality (such as aspects pertaining to the visual presentation of the document that are not dictated by the schema).

According to another exemplary implementation, a method is described for generating an upgrade module for upgrading documents for processing by processing functionality. The method includes: (a) determining whether a particular version of the processing functionality has been created that warrants generation of the upgrade module; and (b) generating the upgrade module if the creation of the particular version warrants the generation of the upgrade module.

Related apparatus and computer readable media are also described herein, as well as additional subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 shows a known technique for upgrading documents produced by a version V1 of an application so that the documents are compatible with a version V2 of the application.

Fig. 2 shows a data processing application including functionality for upgrading arbitrary markup language documents so that these documents are compatible with a current version of the data processing application.

Fig. 3 shows an exemplary architecture of a solution file used, in part, to configure a solution module shown in Fig. 2.

Fig. 4 shows an upgrade generation module used in the data processing application of Fig. 2 to generate an upgrade module.

Fig. 5 shows three exemplary scenarios involving the upgrading of documents.

Fig. 6 shows an exemplary apparatus for implementing the data processing application shown in Fig. 2.

Fig. 7 shows an exemplary user interface (UI) window for designing an electronic form.

Fig. 8 shows another user interface window for modifying the electronic form created using the user interface window shown in Fig. 7.

Fig. 9 shows an overview of a procedure used to upgrade documents.

Figs. 10-12 show additional details regarding the processing steps in the procedure shown in Fig. 9.

Fig. 13 shows the use of an exemplary XSL upgrade module to convert an input XML document into a transformed XML document.

Fig. 14 shows an exemplary computing environment for implementing the data processing application shown in Figs. 2 and 6.

The same numbers are used throughout the disclosure and figures to reference like components and features. Series 100 numbers refer to features originally found in Fig. 1, series 200 numbers refer to features originally found in Fig. 2, series 300 numbers refer to features originally found in Fig. 3, and so on.

DETAILED DESCRIPTION

This disclosure pertains to the visual rendering and editing of structured documents using a data processing application. To provide a concrete framework for discussion, this disclosure will specifically describe the transformation of hierarchically organized data expressed in a markup language into an electronic form that can be visually rendered and edited by an end user. Exemplary forms include a timesheet, work order, travel log, and so on. However, the concepts described herein also have application to other data processing applications besides electronic forms processing.

The terms "upgrade" and "version" have broad connotation as used in this disclosure. "Upgrading" a document refers generally to modifying the document in any manner to suit expectations of a current manifestation (e.g., version) of the document processing application. In one case, an upgrade refers to a scenario in which a user modifies a document produced by an earlier version of a document processing application to be compatible with a later developed, more enhanced, version of the data processing application. However, the term "upgrade" can also refer to modifying a document produced by a later version of the document processing application so that it is compatible with an earlier version of the document processing application. Accordingly, the term "upgrade" as used herein is value-neutral; it could refer to changes that are considered "better" and "worse" than an original version depending on one's particular data processing requirements and objectives. In still a broader interpretation, the term

"upgrade" can refer to the modification of any kind of document produced by any kind of initial document processing application, where such initial document processing application is not considered a predecessor or successor to a current version of a document processing application. And indeed, the initial document need not even be "produced" per se; for instance, the functionality described herein can be used to convert a document that is completely blank or a document that contains completely random information into a format expected by a current version of a document processing application. Due to the liberal interpretation of the term "upgrade," the term "version" should also be understood to have broad connotation as used herein. The term "version" generally refers to any class or kind of document.

The above-noted flexibility of the data processing application highlights one of its principal merits. The data processing application can transform an arbitrary structured document (e.g., a markup language document) so that it conforms to the processing requirements of a current version of a data processing application. In this sense, in one implementation, the data processing application uses a transformation mechanism that is stateless. It is stateless in the sense that it does not require prior knowledge or consideration of the kind of documents that are fed into it.

The above-described property and additional features of the data processing application will be explained in the following disclosure. This disclosure is organized as follows. Section A of this disclosure describes an exemplary design strategy used by a data processing application that includes upgrade functionality. Section B describes an exemplary implementation of the design strategy discussed in Section A. Section C describes an exemplary method of operation of the implementation described in Section B. And Section D describes an exemplary computing environment that can be used to provide the implementation described in Section B.

A. Exemplary Design Strategy

Overview of Exemplary Data Processing Application

Fig. 2 shows an overview of a data processing application 200 for rendering and editing structured documents. To provide a concrete framework for discussion, the upgrading of structured documents is described in the context of the data processing application 200 shown in Fig. 2. However, the upgrading mechanism can be implemented in many different kinds of systems and environments besides the data processing application 200 shown in Fig. 2. Prior to describing the upgrading mechanism itself, the following section describes various features of the data processing application 200.

By way of overview, the data processing application 200 processes structured data 202 expressed in a markup language, transforms this structured data 202 using a solution module 204 to produce transformed information, and presents a rendering of a visual surface 206 on an output device based on the transformed information. An editing user 208 interacts with the visual surface 206, as indicated by arrow 210, using, for instance keyboard 212, mouse device 214, or some other input device. The visual surface 206 can constitute the presentation of a form having data entry fields associated with the structured data 202. In this case, the editing user 208's interaction can involve filling information into the entry fields of the form, such as by entering information into various text boxes, check boxes, etc.

Each of the above-described principal features – structured data 202, solution module 204, and visual surface 206 – will be described in greater detail below.

To begin with, the structured data 202 can be represented in the Extensible Markup Language (XML). XML is a subset of the Standard Generalized Markup

Language (SGML) that enables developers to create customized tags that describe the meaning of data, as opposed to the presentation of data. An XML document is composed of XML elements, each of which includes a start tag (such as <author>), an end tag (such as </author>), and information between the two tags (which is referred to as the content of the elements). An element may include any number (including zero) of name-value pairs (referred to as attributes) related by an equal sign that modifies certain features of the element (such as MONTH = "May"). As shown in Fig. 2, the elements in an XML document have a hierarchical relationship to each other that can be represented as a data tree 216. The elements in the data tree 216 are also commonly referred to as "nodes." A so-called XML schema (not illustrated in Fig. 2) provides a formal specification that defines the types of elements and the organization of elements that should appear in an XML document in order for that document to be considered so-called well formed.

The solution module 204 includes transformation functionality 218. The purpose of the transformation functionality 218 is to transform the structured data 202 into the visual surface 206. The transformation functionality 218 can perform this task using so-called style sheets, such as style sheets provided by Extensible Stylesheet Language Transformation (XSLT). XSLT transforms the XML data into a format appropriate for presentation, such as the Hypertext Markup Language (HTML), Extensible Hypertext Markup Language (XHTML), Dynamic HTML (DHTML), etc. In other words, documents expressed in XML include tags that are particularly tailored to convey the meaning of the data in the documents. The XSLT conversion converts the XML documents into another markup language in which the tags pertain to the visual presentation of the information contained in the documents. (To facilitate discussion, the following description assumes the use of HTML to render the documents; however, other presentation-oriented markup languages can be used to render the documents.) Because

HTML is a markup language, it can be conceptualized as a view tree 220 that includes a hierarchical organization of nodes, as in the case of data tree 216. The reader is referred to the World Wide Web Consortium's (W3C) specifications for background information regarding XML and XSL.

Nodes in the view tree 220 can be mapped (i.e., associated) to corresponding nodes in the data tree 216. Further, nodes in the data tree 216 can be mapped to corresponding nodes the view tree 220. The mapping of nodes in the view tree 220 to nodes in the data tree 216 allows the solution module 204 to correlate editing operations performed on the visual surface 206 to corresponding nodes in an XML document. This allows the solution module 204 to store information entered by the editing user 208 into the XML document during an editing session. Additional information regarding the mapping functionality of the data processing application 200 can be found in the commonly assigned U.S. Patent Application entitled, "Mapping Between Structured Data and a Visual Surface," filed on the same day as the present application, which names Prakash Sikchi, Evgeny N. Veselov, and Stephen J. Mooney as inventors.

The visual surface 206 itself has an appearance that is determined by both the information contained in the structured data 202 as well as the effects of the XSLT transformation provided by the transformation functionality 218. Generally, in the case of electronic forms, the visual surface 206 typically includes a hierarchical structure which is related to the hierarchical structure in the structured data 202. For instance, an exemplary electronic form 222 includes multiple sections pertaining to different topics that reflect the topics in the structured data 202. (However, it is not necessary to have a one-to-one direct correspondence between the organization of the structured data 202 and the organization of the visual surface 206; in other words, the transformation of the structured data 202 to the visual surface 206 is generally considered non-isomorphic).

Each section in the exemplary electronic form 222 can include one or more data entry fields for received input from the editing user 208, such as data entry field 224. The data entry fields are also referred to herein as "editing controls." Different graphical components can be used to implement the editing controls, including text boxes, drop-down list boxes, lists boxes, option buttons (also referred to as radio buttons), check boxes, and so on. Figs 7 and 8, to be described in turn, provides an example of the visual appearance of an electronic form as it is being designed and modified, respectively.

The functionality of the solution module 204 is defined, in part, by a solution file, such as exemplary solution file 226 stored in storage 228. The solution file 226 essentially constitutes an electronic form template, providing all of the semantic information required to transform the structured data 202 into the visual surface 206. Different XML documents may have been created by, or otherwise refer to, different electronic form templates. Accordingly, different XML documents may have different solution files associated therewith. Various techniques can be used to retrieve a solution file that is associated with a particular XML document. For instance, an appropriate solution file can be retrieved based on URN (Uniform Resource Name) or URL (Uniform Resource Locator) information contained in the header of an input XML document. That header information links the input document to a corresponding solution file.

Jumping ahead briefly in the sequence of figures, Fig. 3 shows an exemplary organization of the solution file 226. The solution file 226 itself can include a collection of files (302, 304, 306, 308, and 310) that together provide all of the semantic information required to transform the structured data 202 into the visual surface 206. This collection of files can be packaged together. In one exemplary implementation, this collection of files is referred to using an extension .xsn. A form definition file 302, also called a manifest file, forms the centerpiece of the collection. The form definition file

302 contains information about all of the other files in the solution module 204. This file 302 is assigned the exemplary extension .xsf. A schema file 304 is used to constrain and validate the structured data 302. This file is assigned the exemplary extension .xsd. View files 306 provide presentation logic files that are used to present, view, and transform the structured data 302. These files therefore implement the transformation functionality 218 discussed in connection with Fig. 2. The view files 306 can include multiple files corresponding to multiple possible views (i.e., visual surfaces 206) that the user 208 can select from. The view files 306 are assigned the exemplary extension .xsl. A default data file 308 contains default data that can be displayed in a view for fields that have not been explicitly defined by the user 208. This file 308 is assigned the exemplary extension .xml. Finally, business logic files 310 provide programming code used to implement specific editing behavior, data validation, event handlers, control of data flow, and other features. Such programs can be written in any kind of language, such as the scripting language provided by Microsoft JScript® or VBScript. In this case, these files are assigned the exemplary extensions .js or .vb (for JScript and VBScript, respectively).

With the above introduction, the focus will now turn to the mechanism provided in the data processing application 200 for upgrading documents.

Overview of Upgrade Strategy

Returning to Fig. 2, when the solution module 204 that produces the electronic form 222 was initially created, the design functionality provided by the data processing application 200 would have generated an XSLT file that implemented the necessary mapping between the structured data 202 and the visual surface 206. This XSLT file is referred to as XSLT-V1. Subsequently, assume that a designing user modified the electronic form 222 and associated solution module 204 in a significant way, such as by deleting data entry fields, adding data entry fields, and so on. Such a change would have

the effect of also modifying the schema 304 on which the underlying structured data 202 is based. Accordingly, such a change might warrant changing both the schema 304 associated with the solution module 204 and the XSLT files associated with the solution module 204. As explained above, the schema 304 provides information regarding the general organization and content of structured data 202 as expected by the solution module 204, while the XSLT files provide information regarding how to map the structured data 202 into a desired visual presentation. Fig. 2 specifically illustrates the inclusion of a second version of the XSLT files (i.e., XSLT-V2 230) to handle the above identified change in the electronic form 222.

Fig. 2 shows exemplary documents that can be processed using the version V2 of the solution module 204 (that uses XSLT-V2 230). A first group of documents 232 pertain to documents created using the current version (V2) of the solution module 204. Hence, these documents are referred to as XML-V2 documents (where the suffix V2 indicates that they were created using version V2 of the solution module 204). A second group of documents 234 pertain to documents created using the first version (V1) of the solution module 204. Hence, these documents are referred to as XML-V1 documents. But as noted above, the solution module 204 is not limited to processing documents that were generated by some version of the solution module 204. The solution module 204 can process any kind of documents, including, in extreme cases, a document produced by an application having virtually no relationship to the solution module 204 and the corresponding electronic form 222. Or indeed, a completely blank document or document containing random nodes can be effectively processed by the solution module 204. A group of documents 236 labeled "XML-arbitrary" is included in Fig. 2 to highlight the ability of the data processing application 200 to process completely arbitrary input documents. For the purposes of illustration, Fig. 2 shows that the particular

structured data 202 corresponds to an XML-V1 document produced by a first version of the solution module 204.

An upgrade module 238 allows the solution module 204 to transform the XML-V1 structured data 202 into a format compatible with version V2 of the solution module 204. By way of overview, the upgrade module 238 performs this task by modifying the XML-V1 structured data 202 so that it conforms to the expectations of the V2 solution module 204. Such "expectations" may pertain to schema-type expectations that are specified in the schema 304 of the solution module 204. Other "expectations" may pertain to aspects which are considered important to the visual appearance and/or behavior of the V2 version of the electronic form 222, but might not be dictated by the schema 304 of the solution module 204. The upgrade module 238 is shown as being encompassed by the transformation functionality 218 because it can be implemented as an XSL transformation file. Namely, the upgrade module 238 works by transforming the XML-V1 structured data 202 into another XML document that is compatible with the V2 version of the solution module 204.

An upgrade generation module 240 performs the task of generating the upgrade module 238. The upgrade generation module 240 can be triggered to perform this task when the designing user makes a significant change to the electronic form 222. This will prompt the design aspects of the data processing application 200 to generate an XSL file that will convert an input arbitrary XML document into an XML document that is compatible with the expectations of the new version of the solution module 204. In another implementation, the upgrade generation module 240 can generate the upgrade module 238 automatically in any editing circumstance, that is, regardless of the assessed significance of the change.

Finally, an optional "when-to-upgrade" module 242 determines when to apply the upgrade module 238. The when-to-upgrade module 242 may specify that the upgrade module 238 is to be applied to only documents applied to a range of versions, such as versions 2.0 to 3.3, etc. Information regarding the version of an input document can be gleaned from header information contained in the XML document. This module 242 is optional, however, in the sense that the solution module 204 has the capability of processing any structured document produced by any version of any application, and therefore does not require prior knowledge of the version of the input document.

Advancing to Fig. 3 once again, this figure shows that the upgrade module 238 can be implemented as an XSLT file 306. The XSLT file 306 also includes the XSLT-V2 230 that implements the transformation of the structured data 202 into the visual surface 206. The when-to-apply module 242 can be implemented as part of the form definition files 402. This implementation is exemplary; other data processing applications can implement the upgrading mechanism in different ways than is shown in Figs. 2 and 3.

Fig. 4 illustrates the logic underlying the upgrade generation module 240. As noted above, the upgrade generation module 240 generates the upgrade module 238 in response to the changed expectations of the solution module 204. A set of expectations 402 in Fig. 4 reflects this concept. As noted above, these expectations 402 can be grouped into two categories. Schema expectations 404 refer to expectations that are dictated by the schema 304 of the new version of the form 222. Other expectations 406 pertain to features of the electronic form 222 that are considered important to the designing user, but are not necessarily dictated by the schema 304. Additional details regarding the second category 406 of expectations will be described shortly. In another

implementation, the upgrade generation module 240 can generate the upgrade module 238 in all circumstances.

Fig. 5 shows three different scenarios (502, 504, 506) that illustrate the operation of the upgrade module 238. All three of these scenarios (502, 504, 506) reflect changes made to an input XML document to accommodate expectations specified by the schema 304 of the solution module 204. Once again, the schema 304 specifies the structure of the XML document that the solution module 204 is expecting to receive and process to create the electronic form 222.

To begin with, scenario 502 shows the case where the input document 508 represents a document produced by a first version of the solution module 204, and is therefore labeled XML-V1. XML-V1 508 includes a series of nodes that reflect information contained in the document, including nodes R (i.e., signifying the root), B, C, and D. Schema 510 reflects the schema expectations of the version V2 of the solution module 204. The schema 510 corresponds to the nodes in XML-V1 508, except that a new node A has been added. This new node may reflect a new data entry field added to the electronic form 222 by the designing user. In other words, the new node may reflect a new piece of information that is being collected by the electronic form 222. The upgrade module 238 comes into play by creating a new XML document 512 containing created nodes that reflect the expectations of version V2 of the solution module 204. In the new document 512, the contents of nodes R, B, C, and D are copied from the source document XML-V1 508 into the new document 512. In addition, the upgrade module 238 creates new node A, and may assign a default value to this node.

Scenario 504 represents the case where the document XML-V1 514 contains nodes R, A, B, C, and D, but the schema 516 of version V2 of the solution module 204 omits node A. This change may indicate that the designing user has subsequently

modified the electronic form 222 to delete a data entry field. This change indicates that a piece of information is no longer being collection via the electronic form 222. In this situation, the upgrade module 238 comes into play by creating a new XML document 518 that contains nodes R, B, C, and D having content copied from the same-named nodes in document XML-V1 514, but that fails to create node A (because this node is not specified in the schema 516), and thus effectively deletes node A from the input document XML-V1 514.

Scenario 506 describes the case where the input document XML-V1 520 again includes nodes R, A, B, C, and D. In this case, however, the designing user has renamed node A as node E, and this is reflected in the schema 522. This might correspond to the case where the designing user has renamed a data entry field in the electronic form 222. For example, the designing user might have changed the name of a text entry field from "automobile" to "car," etc. The upgrade module 238 addresses this case by creating nodes R, B, C, and D having content copied from the same-named nodes in the document XML-V1 520. This yields XML document 524. However, because it does not find node A in the schema 522, the upgrade module 238 does not create this node, and thus effectively deletes it. On the other hand, the upgrade module 238 sees the new node E in the schema 522, and thus adds node E to the transformed XML document 524. This is merely one way to handle the case where nodes are renamed.

In other cases, the upgrade module 204 can incorporate functionality having the necessary intelligence to detect that a node in the input XML document reflects an identical node in the schema which has been simply renamed (that is, without having to execute the above-described deleting/adding procedure). For example, the structure of an input document can be compared with the structure of a document expected by a current version of an application (e.g., as reflected by the schema associated with the current

version). The comparison may reveal a close match between a collection of nodes in the input document and a collection of nodes in the expected structure. The comparison may further reveal a positional correspondence between unmatched nodes in the input document and unmatched nodes in the expected structure. In this circumstance, there is a significant probability that these positionally related unmatched nodes reflect nodes that have simply been renamed. A rule can therefore be fashioned to detect the above-noted pattern and interpret it as a node rename. More generally, other kinds of analysis strategies may be used to detect other commonly encountered conversion pattern scenarios. These strategies can generally rely on positional and inferential analysis of node structure and content, as well as a wide variety of other analysis techniques. Some strategies can be fashioned to reflect common patterns and rules found in particular document processing environments. In another implementation, it is also possible to modify the upgrade module 238 such that it is based on some knowledge of the schema of the input XSL-V1 document; however, this solution to the problem has the drawback of making the upgrade module 238 no longer stateless.

The scenarios (502, 504, 506) shown in Fig. 5 are highly simplified examples of the transformation functionality provided by the upgrade module 238. In actual cases, the leaf nodes shown in the trees would likely contain a series of child nodes, which, in turn, might include other respective series of child nodes. In this case, the algorithm described in Fig. 5 is performed recursively to process all of these child nodes. Basically, nodes that are expected by version V2 of the solution module 204, but are missing in the structured data 202, are added by the upgrade module 238, and nodes that are not expected by version V2 of the solution module 204, yet are contained in the structured data 202, are essentially deleted.

The upgrade module 238 can perform other kinds of transformations besides adding and deleting nodes. Some of these other transformations may reflect the dictates of the schema 304 associated with version V2 of the solution module 204. Other transformations may not reflect the dictates of the schema 304; for instance, these other transformations may pertain to visual or behavioral features of the electronic form 222 that the designing user would like duplicated in earlier versions (e.g., V1) of the form 222's documents.

For instance, version V1 of the electronic form 222 might have specified that a certain data entry field was optional. The schema corresponding to version V1 would therefore contain information indicating that a node corresponding to this data entry field was optional. As a result of this optional status, some XML documents produced by version V1 may contain this optional node, while others may not. A subsequent second version V2 of the electronic form 222 might have modified the electronic form 222 to make the optional data entry field mandatory. However, this change might not have been propagated to a corresponding change in the schema file 304. As such, the V2 version of the solution module 204 can still "legally" process documents produced by version V1 of the electronic form 222 that lack the optional node, as these documents still conform to the schema (where this node is indicated as optional). This, however, might result various problems with the visual presentation of the electronic form 222. To prevent this from happening, the upgrade module 238 can also modify an input XML document such that it is conformant with the optional status of various nodes specified in the V2 version of the solution module 204, but not necessarily dictated by the schema 304 of the V2 version of the solution module 204.

In another example, the V2 version of the solution module 204 can capture information regarding the minimum and/or maximum number of nodes of a particular

kind that is expected by the V2 version. This numeric information is referred to as cardinality information. In this case, the upgrade module 238 can be used to enforce this expectation by modifying an input document so that it conforms to the cardinality expectations of the V2 version of the solution module 204.

In another example, the V2 version of the solution module 204 can capture information regarding default values associated with nodes that are expected by the V2 version. In this case, the upgrade module 238 can be used to supply or modify default values in an input document so that this document conforms to the default value expectations of the V2 version of the solution module 204.

In another example, the V2 version of the solution module 204 might specify that certain data entry fields must contain a value – namely, that these data entry field values are mandatory. In this case, the upgrade module 238 can be used to modify the mandatory status of values associated with nodes in the input document so that this document conforms with the mandatory status expectations of the V2 version of the solution module 204.

In another example, the V2 version of the solution module 204 might specify that certain formatting applies data entry fields, such as rich text formatting. Rich text formatting requires the capture of formatting information using a data structure. In this case, the upgrade module 238 can be used to modify the formatting of the input document so that it conforms with selected formatting expectations of the V2 version of the solution module 204. However, in general, changes in formatting are considered relatively minor. Hence, many formatting changes are not "enforced" by the upgrade module 238.

The above-identified expectations are merely illustrative. Different business environments may warrant the use of the upgrade module 238 to enforce a different collection of expectations.

B. Exemplary Apparatus for Implementing Upgrade Technique

Fig. 6 shows an overview of an exemplary apparatus 600 for implementing the data processing application 200 shown in Fig. 1. The apparatus 600 includes a computer 602 that contains one or more processing units 604 and memory 606. Among other information, the memory 606 can store an operating system 608 and the above-described data processing application 200, identified in Fig. 6 as a forms application 610. The forms application 610 can include data files 612 for storing the structured XML data 202, and solution module(s) 614. As noted above, a solution module 614 comprises logic that specifies the appearance and behavior of the visual surface 206 (as was described in connection with Fig. 2). The logic provided by solution module 614 is, in turn, determined, in part, by a solution file (such as a solution file 226 composed of the files shown in Fig. 3). The computer 602 is coupled to a collection of input devices 616, including the keyboard 212, mouse device 214, as well as other input devices 618. The computer 602 is also coupled to a display device 620.

In one exemplary implementation, the forms application 610 includes a design mode and an editing mode. The design mode presents design UI 622 on the display device 620 for interaction with a designing user 624. The editing mode presents editing UI 626 on the display device 620 for interaction with the editing user 208. In the design mode, the forms application 610 creates an electronic form 628, or modifies the structure of the electronic form 628 in a way that affects its basic schema. In other words, the design operation produces the solution module 614 that furnishes the electronic form 628. In the editing mode, the editing user 208 uses the electronic form 628 for its intended purpose – that is, by entering information into the electronic form 628 for a business-related purpose or other purpose.

In the design mode, the forms application 610 can be configured to depict the electronic form 628 under development using a split-screen display technique. More specifically, a forms view portion 630 of the design UI 622 is devoted to a depiction of the normal appearance of the electronic form 628. A data source view portion 632 of the visual surface is devoted to displaying a hierarchical tree 634 that conveys the organization of data fields in the electronic form 628.

Fig. 7 shows design UI that illustrates the allocation of the visual surface 206 into the forms view portion 630 and the data source view portion 632. As described above, the forms view portion 630 contains a depiction of the normal appearance of the form 628 – in this case, exemplary form 702. The form 702 includes a plurality of text box entry fields (e.g., fields 704 and 706). Field 704 is labeled First Name; it allows for entry of an individual's first name. Field 706 is labeled Last Name; it allows for entry of an individual's family name. The data source view portion 632 includes the hierarchical tree 634 showing the nested layout of the text fields (704 and 706) presented in the form 702.

The forms application 610 offers multiple techniques for creating the electronic form 702. According to one technique, the electronic form 702 can be created from scratch by building the electronic form 702 from successively selected editing controls. The exemplary electronic form 702 shown in Fig. 7 is entirely constructed using the text entry boxes (704 and 706), but other electronic forms can include other kinds of entry fields (i.e., editing controls), such as drop-down list boxes, list boxes, option button, check boxes, and so on.

Once a form has been created, its design (and associated schema) can be further modified. Fig. 8 shows one exemplary technique for performing this operation. In this technique, the designing user 624 can activate UI functionality 802. This functionality 802 allows the designing user 624 to modify the electronic form 702 that was previously

created (in Fig. 7), to thereby produce new electronic form 804. There are a myriad of ways that the designing user 624 can decide to modify a prior electronic form, such as by adding data entry fields, deleting existing data entry fields, renaming data entry fields, changing the type of editing control assigned to the data entry fields, changing the layout of data entry fields or other features of the electronic form, or making changes to other properties of the electronic form. In the exemplary case shown in Fig. 8, the designing user 624 has decided to add a new data entry field 806 to the electronic form 804 (for entering an individual's age). If the designing user 624 makes changes to the form view portion 630, then corresponding changes will be reflected in the data source view 632. In the present scenario, this is reflected by the introduction of the additional node 808 in the data source view 632.

The changes illustrated in Fig. 8 will warrant making a change to the schema 304 of the solution module 204 associated with this form 804. These changes will also warrant making changes to the XSLT used to create the visual appearance of the electronic form 804. But other changes made to the form 804 might not require modification of the schema 304, as noted in Section A of this disclosure.

In summary, form 702 shown in Fig. 7 can be referred to as version V1 having a corresponding V1 solution module 204. Form 804 shown in Fig. 8 can be referred to as version V2 having a corresponding V2 solution module 204. Accordingly, documents created using the electronic form 702 can be called XML-V1 documents. Documents created using the electronic form 804 can be called XML-V2 documents. By virtue of the changes shown in Fig. 8, the forms application 610 also produces an upgrade module 238. The upgrade module 238 is specifically adapted to transform an arbitrary XML document into an XML document that conforms to expectations associated with electronic form 804 and its corresponding V2 solution module 204.

More specifically, suppose that the editing user 208 makes a request to display and/or edit a particular XML-V1 document. In one implementation, this would prompt the forms application 610 to locate the solution module 204 corresponding to the version V2 of the solution module 204, as this is the most current version. More specifically, the XML-V1 document contains information that identifies a particular kind of form in its header. The forms application 610 can be configured to retrieve the most current version of the solution module 204 corresponding to the kind of form identified in the XML-V1 document's header.

Having retrieved the V2 solution module 204, the forms application 610 proceeds to transform the XML-V1 document into an XML document which conforms to the expectations of the V2 solution module 204. The following flow charts provide additional details regarding the operations performed in generating and applying the upgrade module 238.

C. Exemplary Method of Operation

Overview of Procedure

Fig. 9 shows an overview of a procedure 900 for creating and applying the version upgrade module 238. To begin with, step 902 entails creating version V1 of a solution module 204. To provide a concrete example, this step 902 can correspond to the creation of the electronic form 702 shown in Fig. 7, which is accompanied by the creation of a corresponding V1 solution module 204. Step 904 entails creating an XML document using the electronic form 702 and associated V1 solution module 204. Since this document is created with version V1 of the electronic form 702, it is referred to as XML-V1.

In step 906, the designing editing user 624 creates a version V2 of the electronic form, corresponding, in one example, to the electronic form 804 shown in Fig. 8. This step 906 can entail adding data entry fields to the electronic form 702, deleting data entry fields, changing the properties of existing data entry fields (such as changing the optional status of data entry fields), and so on. The changes made by the designing user 624 may be significant or relatively minor. If the changes are deemed significant, then step 908 entails generating an upgrade module 238 associated with the changes. If the changes are deemed relatively minor, the forms application 610 will not generate an upgrade module. As previously described, a significant change in an electronic form may pertain to a change in the form's schema, or may pertain to an important aspect of its visual appearance or behavior that is not necessarily dictated by the schema. A relatively minor change might reflect a small change to the appearance of the form, such as a change in the formatting applied to the form. Generally, the decision of when to generate the upgrade module 238 can be tailored to address the requirements and objectives of a particular business environment. Fig. 10 provides additional details regarding the generation of the upgrade module 238. In another implementation, an upgrade module 238 is generated in all circumstances, e.g., even for relatively minor changes.

In step 910, the editing user 208 attempts to display the XML-V1 document. In step 912, the forms application 610 determines what technique to use to display the XML-V1 document. In one case, the forms application 610 retrieves the version V2 of the solution module 204 corresponding to form 802 shown in Fig. 8. In another case, the forms application 610 will apply a custom script to display the XML-V1 document, or will use some other technique to render the XML-V1 document. Fig. 11 provides additional details regarding the selection of a technique used to render the XML-V1 document.

Finally, in step 914, the forms application 610 displays the XML-V1 document using the technique determined in step 912. Fig. 12 provides additional details regarding a procedure used to display the XML-V1 document by applying the upgrade module 238 provided by the V2 solution module 204.

Generation of the Upgrade Module

Fig. 10 shows additional details regarding the procedure 908 used to generate the upgrade module 238 (where procedure 908 corresponds to the step 908 shown in Fig. 9). In step 1002, the forms application 610 determines whether the designing user 624 has made a change to the electronic form (e.g., the electronic form 702) that warrants generating an upgrade module 238. Once again, a change that affects the schema 304 of a solution module 204 would generally warrant the generation of an upgrade module 238. Other changes may trigger the generation of an upgrade module 238 even though they do not affect the schema 304 of the solution module 204. In step 1004, if the changes are deemed significant, then the forms application 610 generates the upgrade module 238. In step 1006, if the changes are deemed to be minor, then no upgrade module 238 is generated. As stated above, this functionality can be omitted from the forms application 610. For instance, the forms application 610 can be configured to generate the upgrade module 238 in all circumstances, or based on some other criteria than that described above.

Selection of Document Processing Technique

Fig. 11 shows additional details regarding the procedure 912 used to select a technique for processing the XML-V1 document (where procedure 912 corresponds to the step 912 shown in Fig. 9).

In step 1102, the forms application 610 determines whether it should display the XML-V1 document using an upgrade module 238 associated with a later version of the

electronic form, such as version V2 of the electronic form. More specifically, in one implementation, the XML-V1 document has header information that specifies a solution module 204 that should be used to render the document. In response to a user's request to render the XML-V1 document, the forms application 610 retrieves an appropriate solution module 204 based on the header information in the XML-V1 document. The retrieved solution module 204 contains an upgrade module 238 as well as the when-to-apply module 242. The when-to-apply module 242 determines whether to apply the upgrade module 238 to the input document, such as, in this case, the XML-V1 document. An exemplary excerpt of instructions provided by the when-to-apply module 242 is as follows:

```
<xsf:documentVersionUpgrade>
        <xsf:useTransform transform="upgrade.xsl"
    minVersionToUpgrade="1.0" maxVersionToUpgrade="1.9">
    </xsf:useTransform>
</xsf:documentVersionUpgrade>
```

These instructions state that the upgrade module 238 is to be applied to versions within a specified range, i.e., between minVersionToUpgrade and maxVersionToUpgrade. In this case, the minVersionToUpgrade is identified as version 1.0, and the maxVersionToUpgrade is specified as version 1.9. Since the user is attempting to render a document (XML-V1) produced by version V1, then the forms application 610 will apply the upgrade module 238 to this version. Step 914 involves displaying the XML-V1 document using the upgrade module 238. Details of step 914 are presented in Fig. 12.

In addition, the forms application 610 provides other techniques for processing the XML-V1 document. For instance, in step 1106, the forms application 610 determines whether the retrieved solution module 204 has been configured to apply a special script to convert the XML-V1 document into a format compatible with the retrieved solution module 204. Such a script can be created by a designing user and stored in the business logic 310 of the solution module 204. The forms application 610 can automatically access and apply the script in response to an editing user's request to edit the XML-V1 input document. Step 1108 represents the application of the custom script to the XML-V1 document.

In another case, as reflected in step 1110, the forms application 610 can apply a custom XSL upgrade module to the inputted document (XML-V1), rather than the upgrade module 238 that was automatically generated by the forms application 610 when the designing user 624 made a change in the electronic form. If this technique is activated, in step 1112, the forms application 610 applies the custom XSL module to the input document.

In another case, as reflected in step 1114, the forms application 610 can process the XML-V1 document without first transforming this document using any upgrade module. In this case, as reflected in step 1116, the XML-V1 document is displayed as if it was a document created by a later version (e.g., version V2), which is not the case. This might result in one or more anomalies in the visual presentation of the electronic form, and one or more anomalies in the behavior of the thus displayed form.

Finally, step 1118 indicates that the forms application 610 can render the XML-V1 document using still additional techniques. Generally, as was assumed in the above discussion, the forms application 610 can be configured to automatically select any of the options shown in Fig. 11. This can be performed by reading and applying selection

preferences specified by the user in advance. In another implementation, the forms application 610 can provide a menu at runtime which identifies the various document processing options available to the editing user 208. This gives the editing user 208 the option of selecting a strategy at that time.

The Operation of the Upgrade Module

Fig. 12 describes a procedure 914 used to render a document using an upgrade module 238. This procedure corresponds to step 914 shown in Figs. 9 and 11. To provide a concrete example, the procedure 914 will be described in the context of the scenario illustrated in Figs. 7 and 8. In this scenario, version V1 of the electronic form 702 includes nodes corresponding to First Name and Last name (associated with data entry fields 704 and 706, respectively). The electronic form 702 is modified as shown in Fig. 8 to produce version V2 of the electronic form 804 that contains the additional node related to Age (associated with data entry field 806). In the case shown in Fig. 12, the editing user 208 is seeking to render a document produced by version V1 of the electronic form 702 (i.e., an XML-V1 document) using version V2 of the solution module 204.

More specifically, the procedure 914 focuses on an exemplary routine 1202 for processing a single leaf node that is expected by version V2 of the solution module 204. As indicated by loop 1204, the routine 1202 is repeated a plurality of times for individual nodes expected by version V2 of the solution module 204. That is, for instance, some of the nodes expected by version V2 include a collection of child nodes associated therewith. Also, these child nodes may have their own respective child nodes. Nodes that have a plurality of nodes associated therewith are referred to as container nodes. The loop 1204 generally indicates that the routine 1202 is repeated for individual nodes within a container.

Generally, routine 1202 processes an XML-V1 document by making reference to the expectations of version V2 of the solution module 204 to produce a modified document. For a single expected node, step 1206 of the routine 1202 creates a corresponding node in the modified document. That is, if the schema of the version V2 of the solution module 204 indicates that a node is expected, then the routine 1202 creates this node. A node that is present in the XML-V1 document that is not expected by the V2 schema is not created, and thus is effectively deleted.

Step 1208 determines whether the node created in step 1206 has a counterpart node in the XML-V1 document. If so, in step 1210, the routine 1202 copies the content of the counterpart XML-V1 node into the node created in step 1206. If step 1208 is answered in the negative, then a step 1212 can add default content to the node created in step 1206. In other words, assume that the XML-V1 document omits a node that is expected by the V2 version. In this case, the routine 1202 will add that node to the modified document in step 1206, and then assign a default value to that node in step 1212.

Once again, the loop 1204 indicates that the routine 1202 is repeated a plurality of times for additional nodes expected by version V2 of the solution module 204.

As a result of the above-described processing, an XML-V1 document can be successfully processed using the XSLT associated with version V2 of the solution module 204. In effect, the upgrade module 238 internally transforms the XML-V1 document into an XML-V2 document. The same procedure can be performed to transform any input document, such as, in one case, an input document that has a completely arbitrary form relative to the schema of the version V2.

In the exemplary context of Figs. 7 and 8, the upgrade procedure has the effect of adding an age node to the XML-V1 document before it is transformed by the XSLT

associated with version V2 of the solution module 204. Fig. 13 illustrates the transformation produced by the upgrade module in this case. In that figure, an input XML-V1 document 1302 is input to a V2 solution module 204. The input document 1302 does not contain a node for Age. In response, the V2 solution module 204 applies the XSL upgrade module 1304. Instructions 1306 in the upgrade module 1304 are specifically responsible for adding an age node to the input XML-V1 document 1302. The upgrade module 1304 has the end effect of producing XML-transformed document 1308. The resultant XML-transformed document 1308 contains an element 1310 pertinent to the added age node. This new node is also assigned the default age of "18," as specified by the upgrade module 1304.

The procedure 914 shown in Figs. 12 and 13 emphasized the processing of added and deleted nodes. These are changes that are dictated by the schema 304 of version V2 of the solution module 204. However, the upgrade module 238 can also transform input documents so that they satisfy other expectations associated with the electronic form 222 that are not necessarily dictated by its schema 304. These other aspects might pertain to the visual presentation of the electronic form 222 or its editing behavior, and so on.

D. Exemplary Computer Environment

Fig. 14 illustrates one example of a computing environment 1400 within which the above-described forms application 610 can be either fully or partially implemented. The computing environment 1400 includes the general purpose computer 602 and display device 620 discussed in the context of Fig. 6. However, the computing environment 1400 can include other kinds of computer and network architectures. For example, although not shown, the computer environment 1400 can include hand-held or laptop devices, set top boxes, programmable consumer electronics, mainframe computers, gaming consoles,

etc. Further, Fig. 14 shows elements of the computer environment 1400 grouped together to facilitate discussion. However, the computing environment 1400 can employ a distributed processing configuration. In a distributed computing environment, computing resources can be physically dispersed throughout the environment.

Exemplary computer 602 includes one or more processors or processing units 604, a system memory 606, and a bus 1402. The bus 1402 connects various system components together. For instance, the bus 1402 connects the processor 604 to the system memory 606. The bus 1402 can be implemented using any kind of bus structure or combination of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. For example, such architectures can include an Industry Standard Architecture (ISA) bus, a Micro Channel Architecture (MCA) bus, an Enhanced ISA (EISA) bus, a Video Electronics Standards Association (VESA) local bus, and a Peripheral Component Interconnects (PCI) bus also known as a Mezzanine bus.

Computer 602 can also include a variety of computer readable media, including a variety of types of volatile and non-volatile media, each of which can be removable or non-removable. For example, system memory 606 includes computer readable media in the form of volatile memory, such as random access memory (RAM) 1404, and non-volatile memory, such as read only memory (ROM) 1406. ROM 1406 includes an input/output system (BIOS) 1408 that contains the basic routines that help to transfer information between elements within computer 602, such as during start-up. RAM 1404 typically contains data and/or program modules in a form that can be quickly accessed by processing unit 604.

Other kinds of computer storage media include a hard disk drive 1410 for reading from and writing to a non-removable, non-volatile magnetic media, a magnetic disk drive

1412 for reading from and writing to a removable, non-volatile magnetic disk 1414 (e.g., a "floppy disk"), and an optical disk drive 1416 for reading from and/or writing to a removable, non-volatile optical disk 1418 such as a CD-ROM, DVD-ROM, or other optical media. The hard disk drive 1410, magnetic disk drive 1412, and optical disk drive 1416 are each connected to the system bus 1402 by one or more data media interfaces 1420. Alternatively, the hard disk drive 1410, magnetic disk drive 1412, and optical disk drive 1416 can be connected to the system bus 1402 by a SCSI interface (not shown), or other coupling mechanism. Although not shown, the computer 602 can include other types of computer readable media, such as magnetic cassettes or other magnetic storage devices, flash memory cards, CD-ROM, digital versatile disks (DVD) or other optical storage, electrically erasable programmable read-only memory (EEPROM), etc.

Generally, the above-identified computer readable media provide non-volatile storage of computer readable instructions, data structures, program modules, and other data for use by computer 602. For instance, the readable media can store the operating system 608, one or more application programs 1422 (such as the forms application 610), other program modules 1424, and program data 1426.

The computer environment 1400 can include a variety of input devices. For instance, the computer environment 1400 includes the keyboard 212 and a pointing device 214 (e.g., a "mouse") for entering commands and information into computer 602. The computer environment 1400 can include other input devices (not illustrated), such as a microphone, joystick, game pad, satellite dish, serial port, scanner, card reading devices, digital or video camera, etc. Input/output interfaces 1428 couple the input devices to the processing unit 604. More generally, input devices can be coupled to the computer 602 through any kind of interface and bus structures, such as a parallel port, serial port, game port, universal serial bus (USB) port, etc.

The computer environment 1400 also includes the display device 620. A video adapter 1430 couples the display device 620 to the bus 1402. In addition to the display device 620, the computer environment 1400 can include other output peripheral devices, such as speakers (not shown), a printer (not shown), etc.

Computer 602 can operate in a networked environment using logical connections to one or more remote computers, such as a remote computing device 1432. The remote computing device 1432 can comprise any kind of computer equipment, including a general purpose personal computer, portable computer, a server, a router, a network computer, a peer device or other common network node, etc. Remote computing device 1432 can include all of the features discussed above with respect to computer 602, or some subset thereof.

Any type of network can be used to couple the computer 602 with remote computing device 1432, such as a local area network (LAN) 1434, or a wide area network (WAN) 1436 (such as the Internet). When implemented in a LAN networking environment, the computer 602 connects to local network 1434 via a network interface or adapter 1438. When implemented in a WAN networking environment, the computer 602 can connect to the WAN 1436 via a modem 1440 or other connection strategy. The modem 1440 can be located internal or external to computer 602, and can be connected to the bus 1402 via serial I/O interfaces 1442 other appropriate coupling mechanism. Although not illustrated, the computing environment 1400 can provide wireless communication functionality for connecting computer 602 with remote computing device 1432 (e.g., via modulated radio signals, modulated infrared signals, etc.).

In a networked environment, the computer 602 can draw from program modules stored in a remote memory storage device 1444. Generally, the depiction of program modules as discrete blocks in Fig. 14 serves only to facilitate discussion; in actuality, the

programs modules can be distributed over the computing environment 1400, and this distribution can change in a dynamic fashion as the modules are executed by the processing unit 604.

Wherever physically stored, one or more memory modules 606, 1414, 1418, 1444, etc. can be provided to store the forms application 610.

Although the invention has been described in language specific to structural features and/or methodological acts, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or acts described. Rather, the specific features and acts are disclosed as exemplary forms of implementing the claimed invention.